Binary Heap

Bruce Nan

Motivation

- Development of a data structure which allows efficient inserts and efficient deletes of the minimum value (minheap) or maximum value (maxheap)

Heap

- Definition in Data Structure
  - Heap: A special form of complete binary tree that key value of each node is no smaller (larger) than the key value of its children (if any).

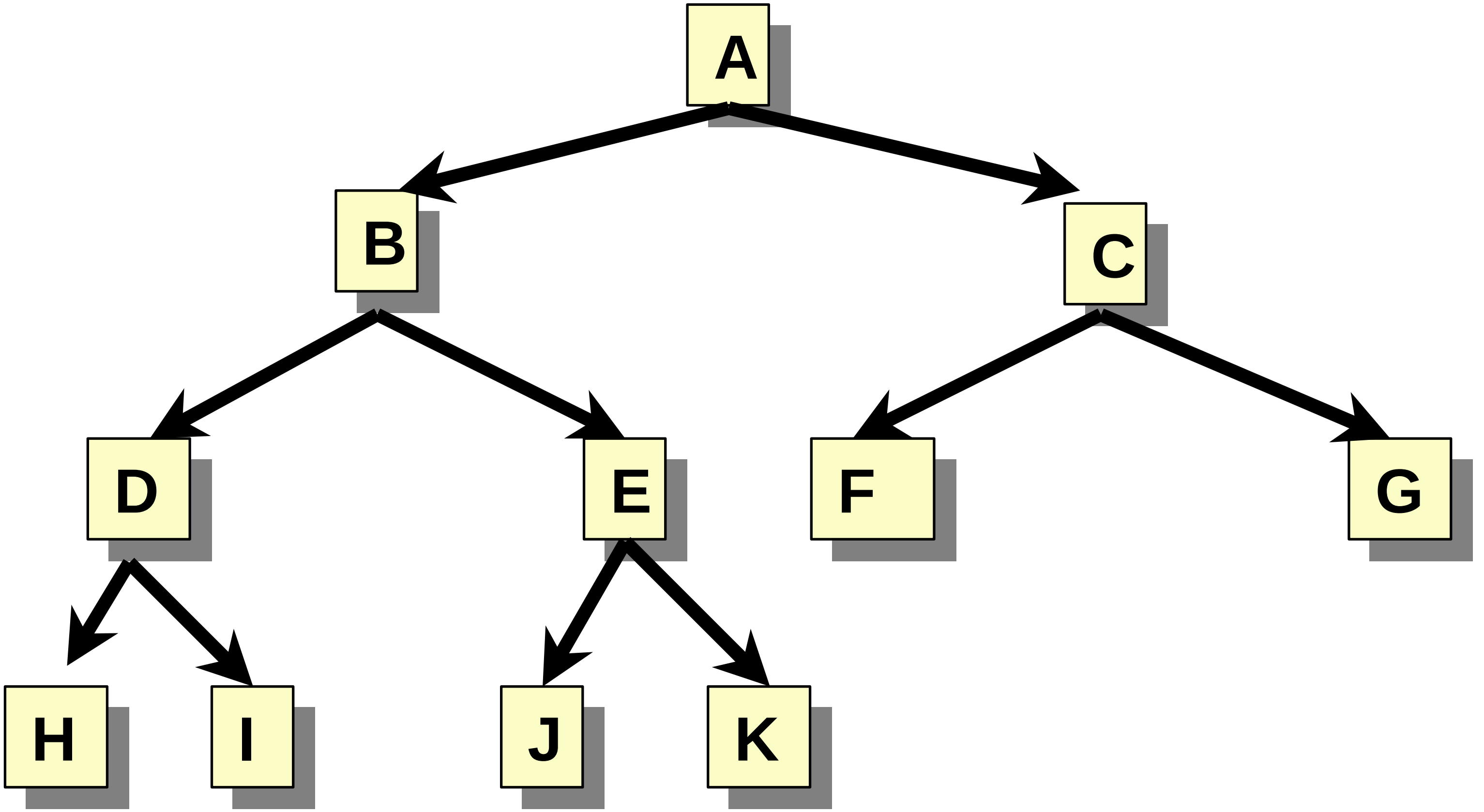
Max (Min) Heap

- Max-Heap: root node has the largest key.
  - A *max tree* is a tree in which the key value in each node is no smaller than the key values in its children. A *max heap* is a complete binary tree that is also a max tree.
- Min-Heap: root node has the smallest key.
  - A *min tree* is a tree in which the key value in each node is no larger than the key values in its children. A *min heap* is a complete binary tree that is also a min tree.

Complete Binary Tree

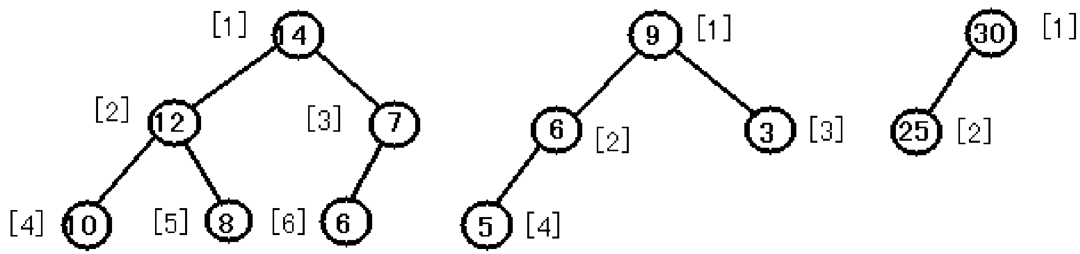
– A **complete binary tree** is a binary tree in which every level, *except possibly the last*, is completely filled, and all nodes are as far left as possible.

Complete Binary Trees - Example

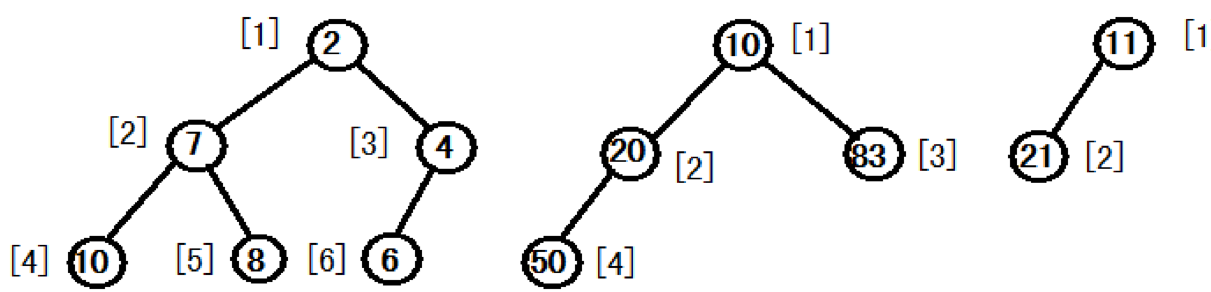


Heap

- Example:
  - Max-Heap

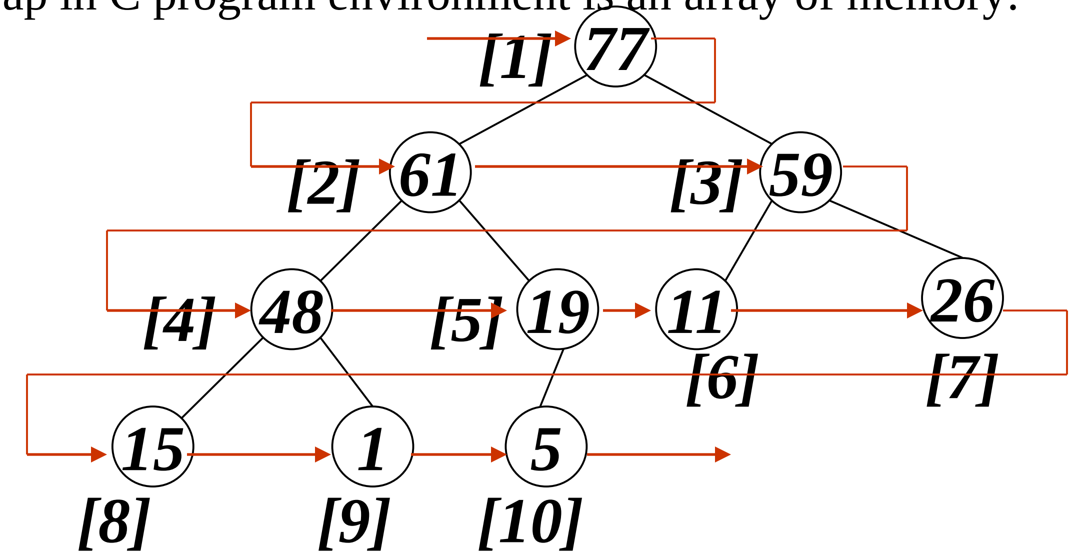


  - Min-Heap



Heap

- Notice:
  - Heap in data structure is a complete binary tree! (**Nice representation in Array**)
  - Heap in C program environment is an array of memory.



  - Stored using array in C

index	1	2	3	4	5	6	7	8	9	10
value	77	61	59	48	19	11	26	15	1	5

Heap

- Operations
  - Creation of an empty heap
  - Insertion of a new element into the heap
  - Deletion of the largest(smallest) element from the heap
- Heap is complete binary tree, can be represented by array. So the complexity of inserting any node or deleting the root node from Heap is O(height) = $O(\log_2 n)$.

Implementation

- One-array implementation of a binary tree
- Root of tree is at element 1 of the array
- If a node is at element i of the array, then its children are at elements $2*i$ and $2*i+1$
- If a node is at element i of the array, then its parent is at element $floor(i/2) = \lfloor i/2 \rfloor$

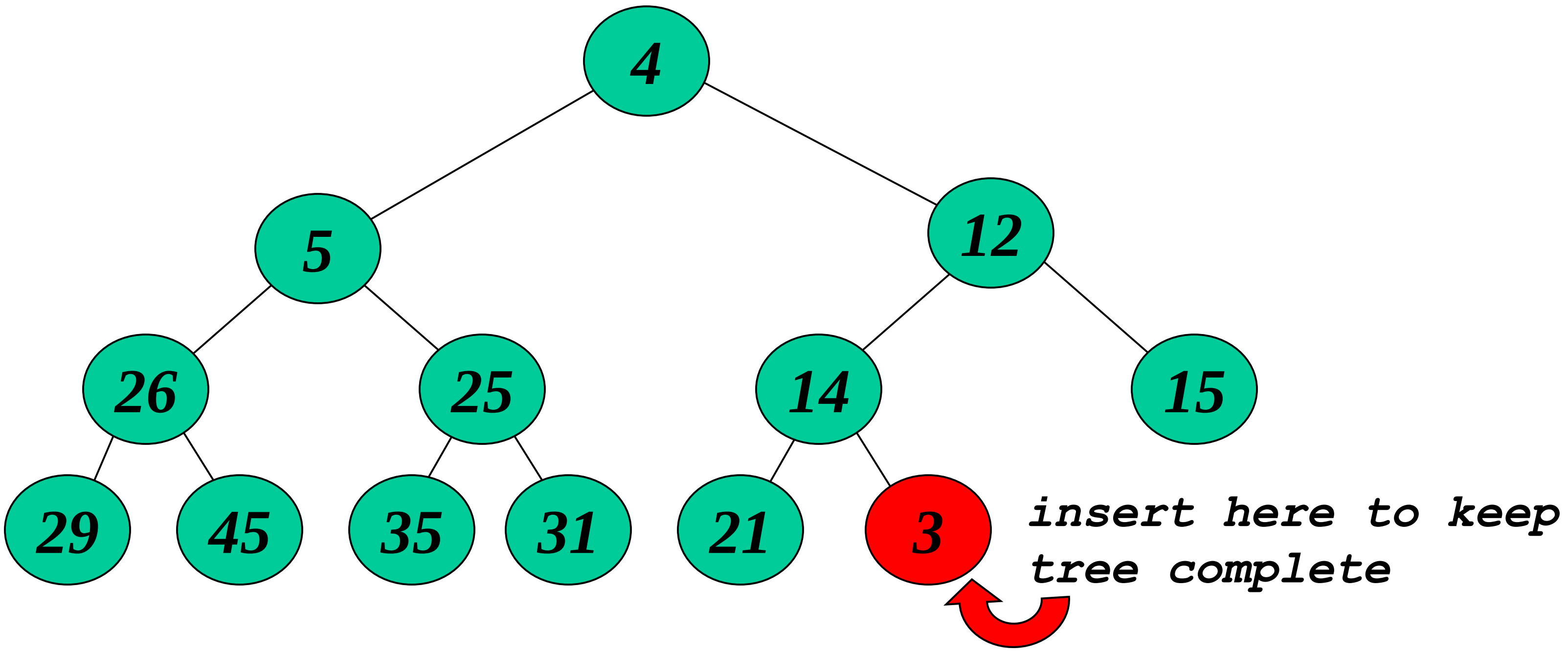
Heap

- Given the index i of a node
- Parent(i)
  – return i/2
- LeftChild(i)
  – return 2i
- RightChild(i)
  – Return 2i+1

Insertion into a Max Heap

```
void insert_max_heap(element item, int &n)
{
  int i;
  if (HEAP_FULL(n)) {
    fprintf(stderr, "the heap is full.\n");
    exit(1);
  }
  i = ++n;
  while ((i!=1)&&(item.key>heap[i/2].key)) {
    heap[i] = heap[i/2];
    i /= 2;
  }
  heap[i]= item;
}
```

Inserting a Value to a Min Heap

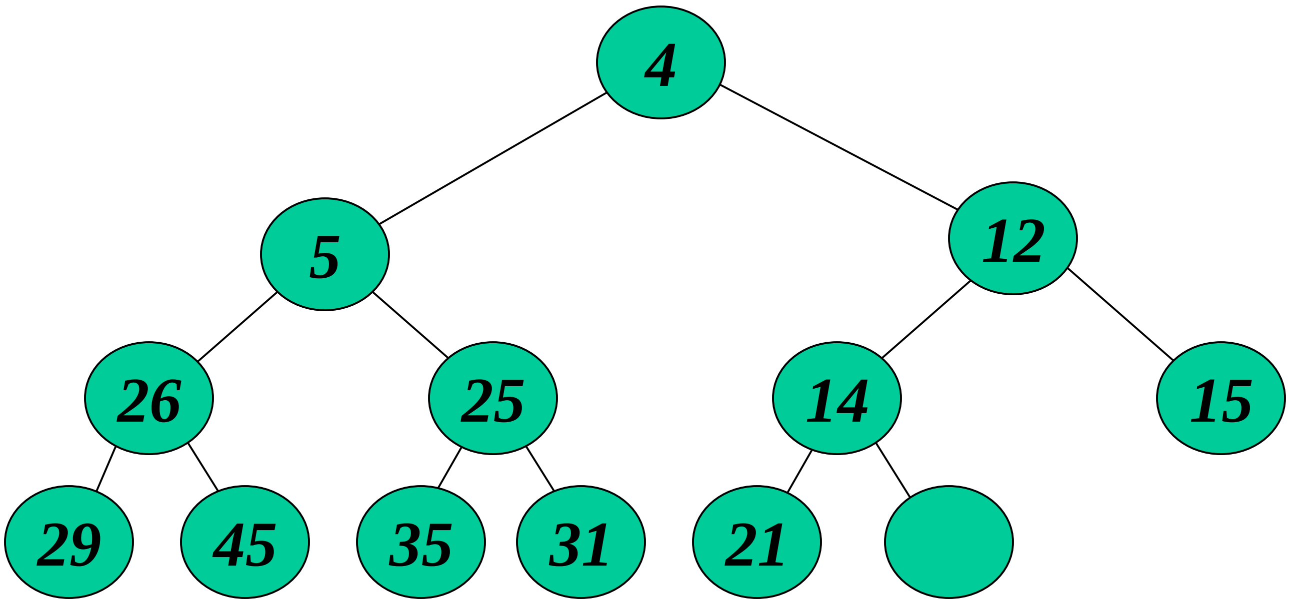


i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
array	__	4	5	12	26	25	14	15	29	45	35	31	21	3	__	__

currentsize = 13

Insert 3

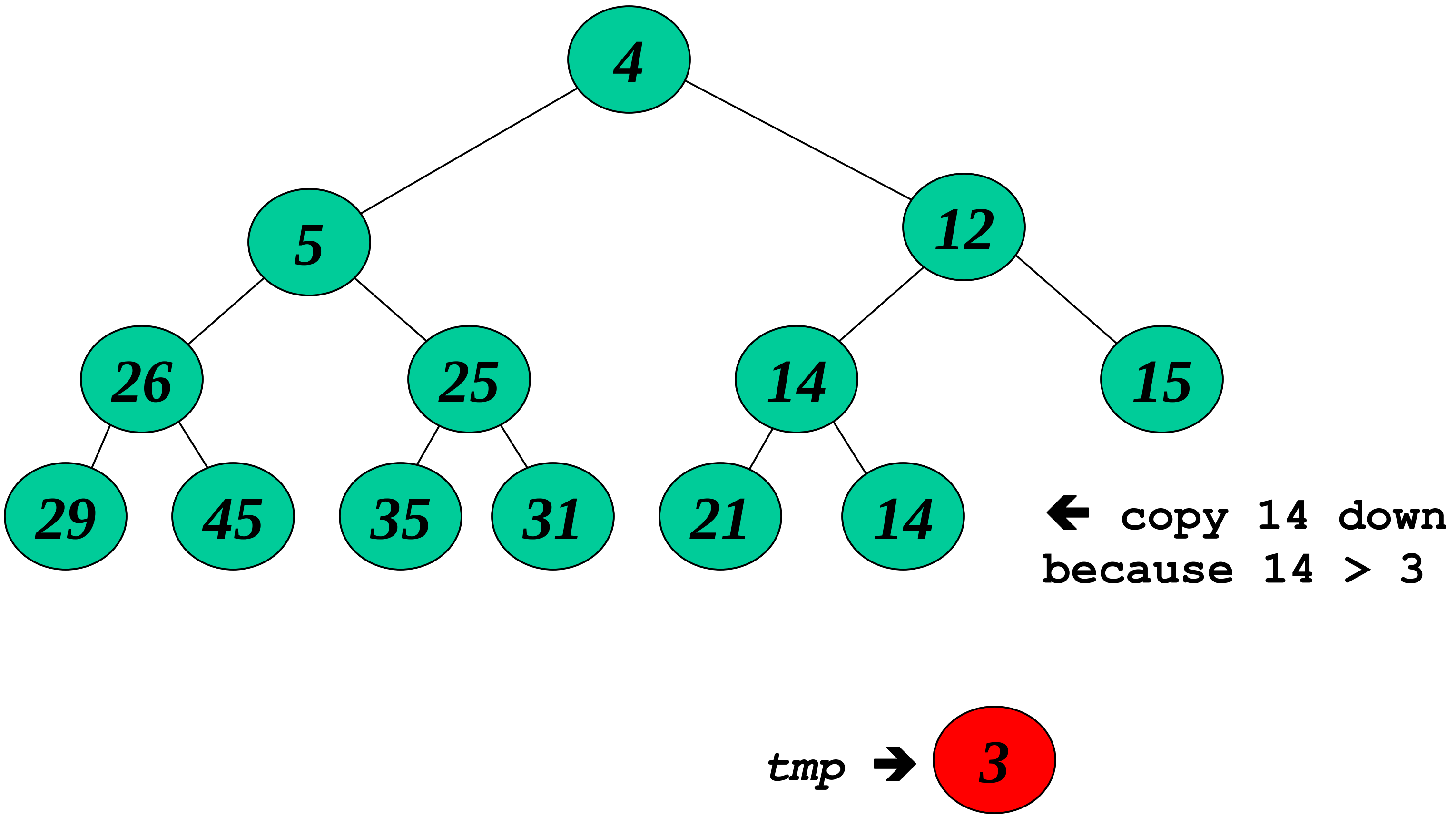
Inserting a Value



save new value in a temporary location: tmp →

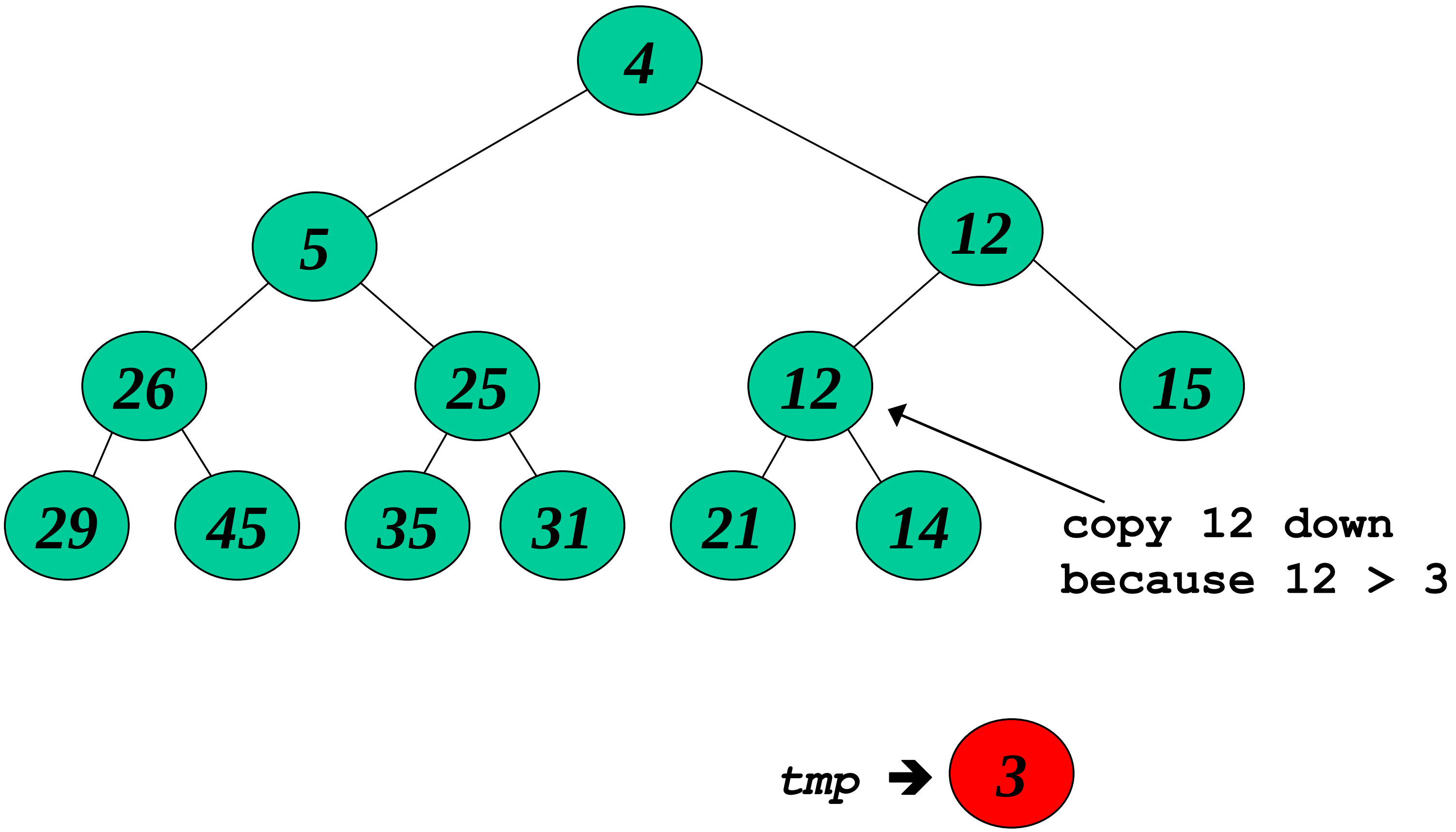
Insert 3

Inserting a Value



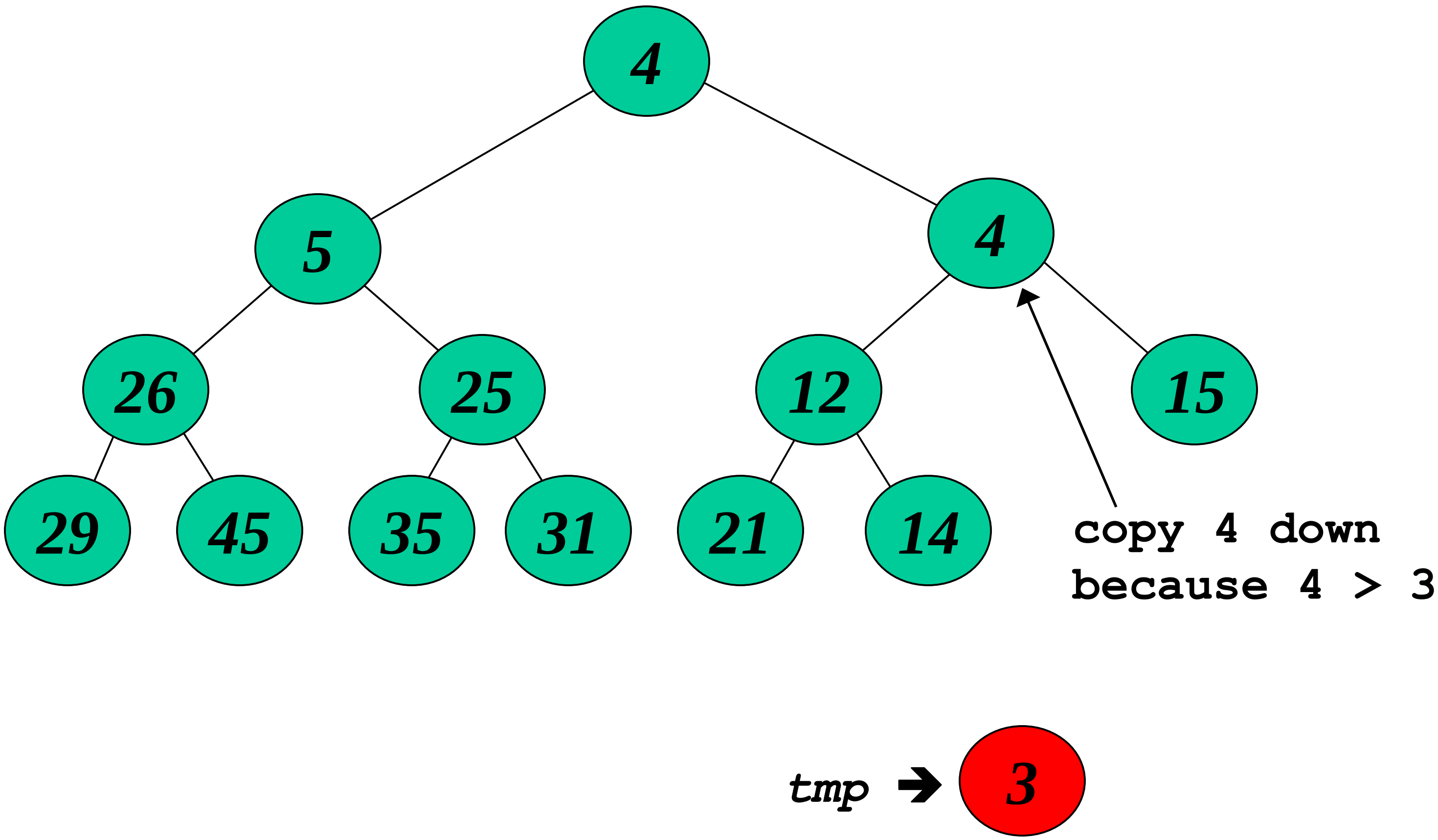
Insert 3

Inserting a Value



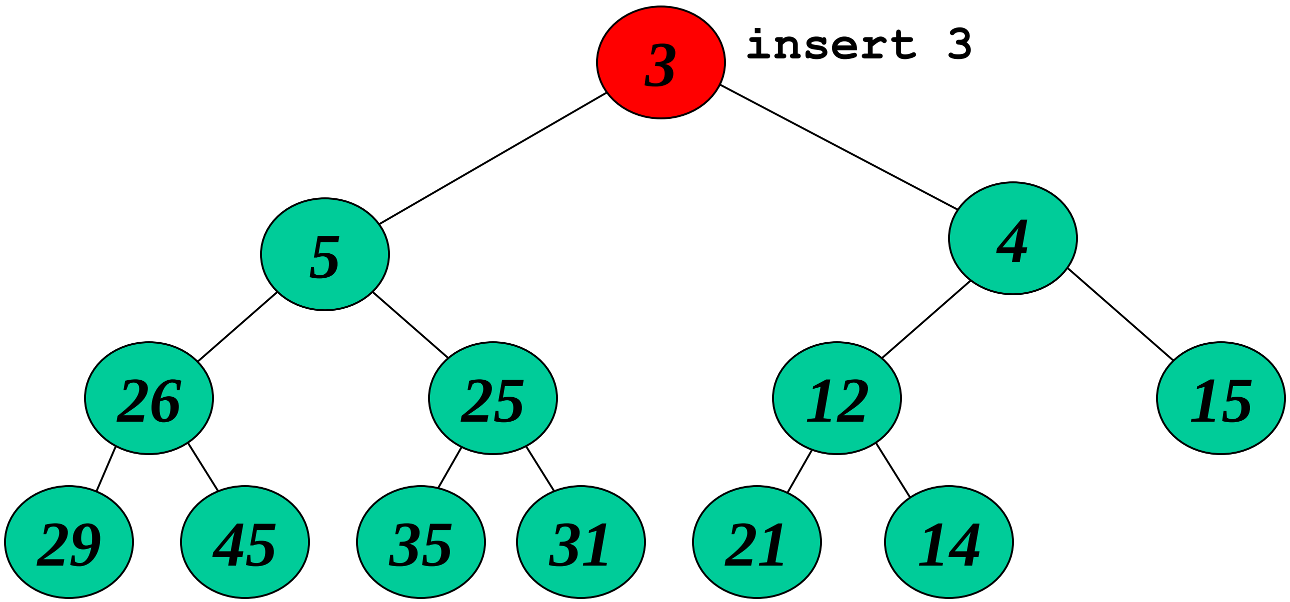
Insert 3

Inserting a Value



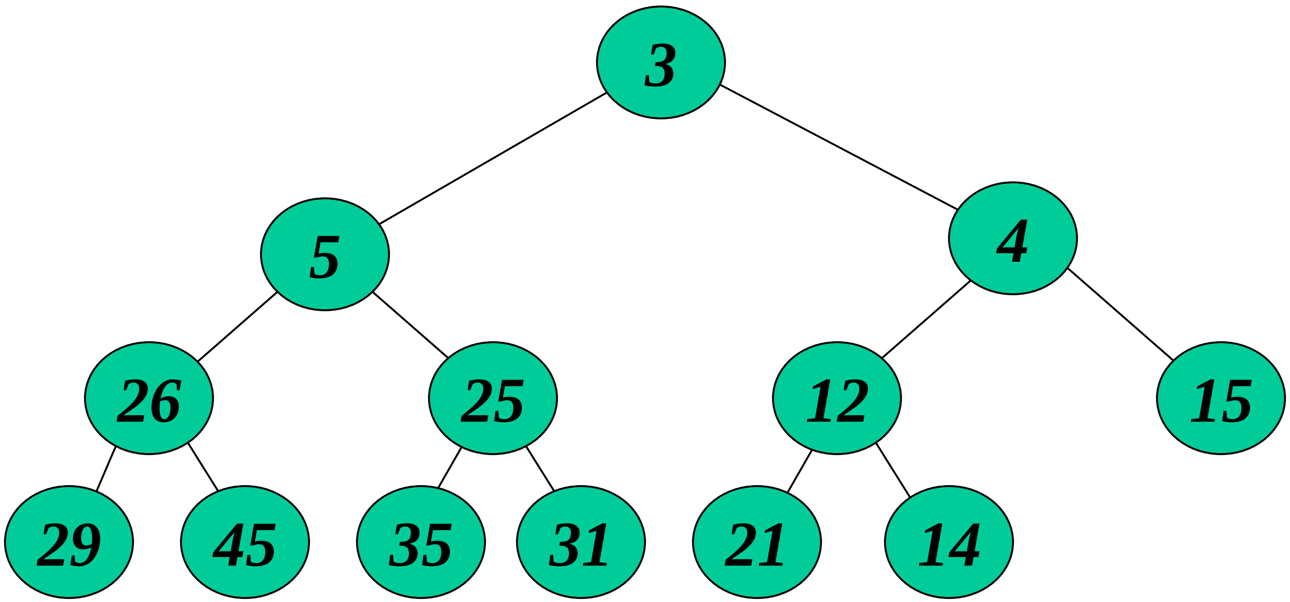
Insert 3

Inserting a Value



Insert 3

Heap After Insert



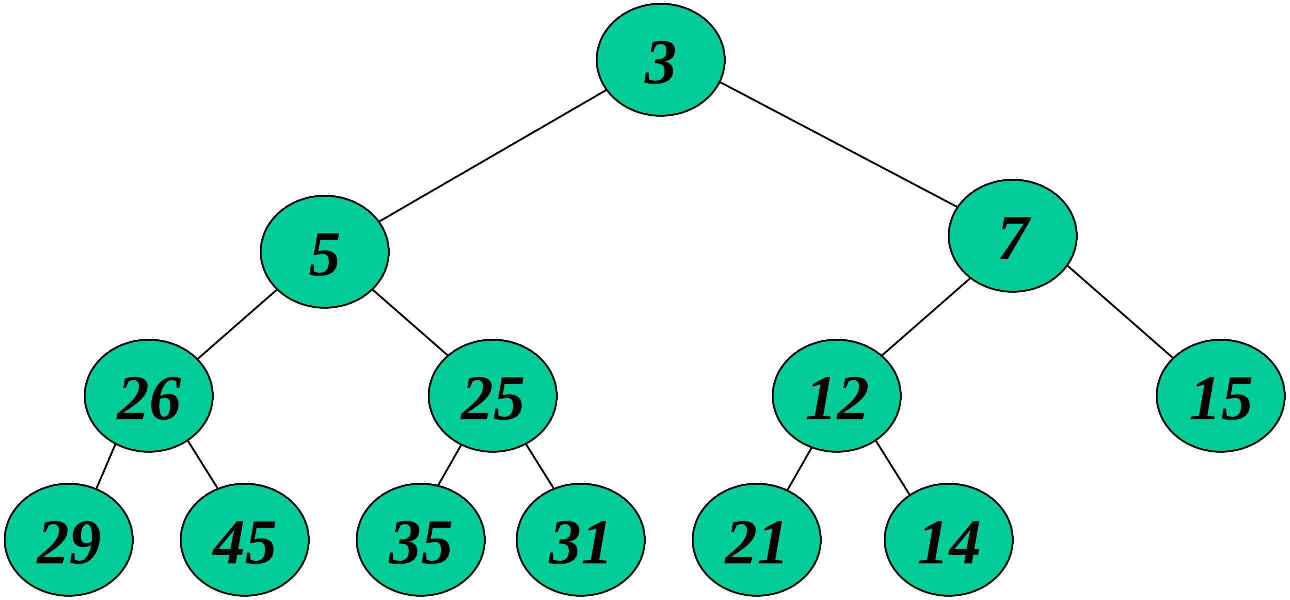
Deletion from a Max Heap

```
element delete_max_heap(int &n)
{
  int parent, child;
  element item, temp;
  if (HEAP_EMPTY(n)) {
    fprintf(stderr, "The heap is empty\n");
    exit(1);
  }
  /* save value of the element with the
     highest key */
  item = heap[1];
  /* use last element in heap to adjust heap */
  temp = heap[n--];
  parent = 1;
  child = 2;
```

Deletion from a Max Heap

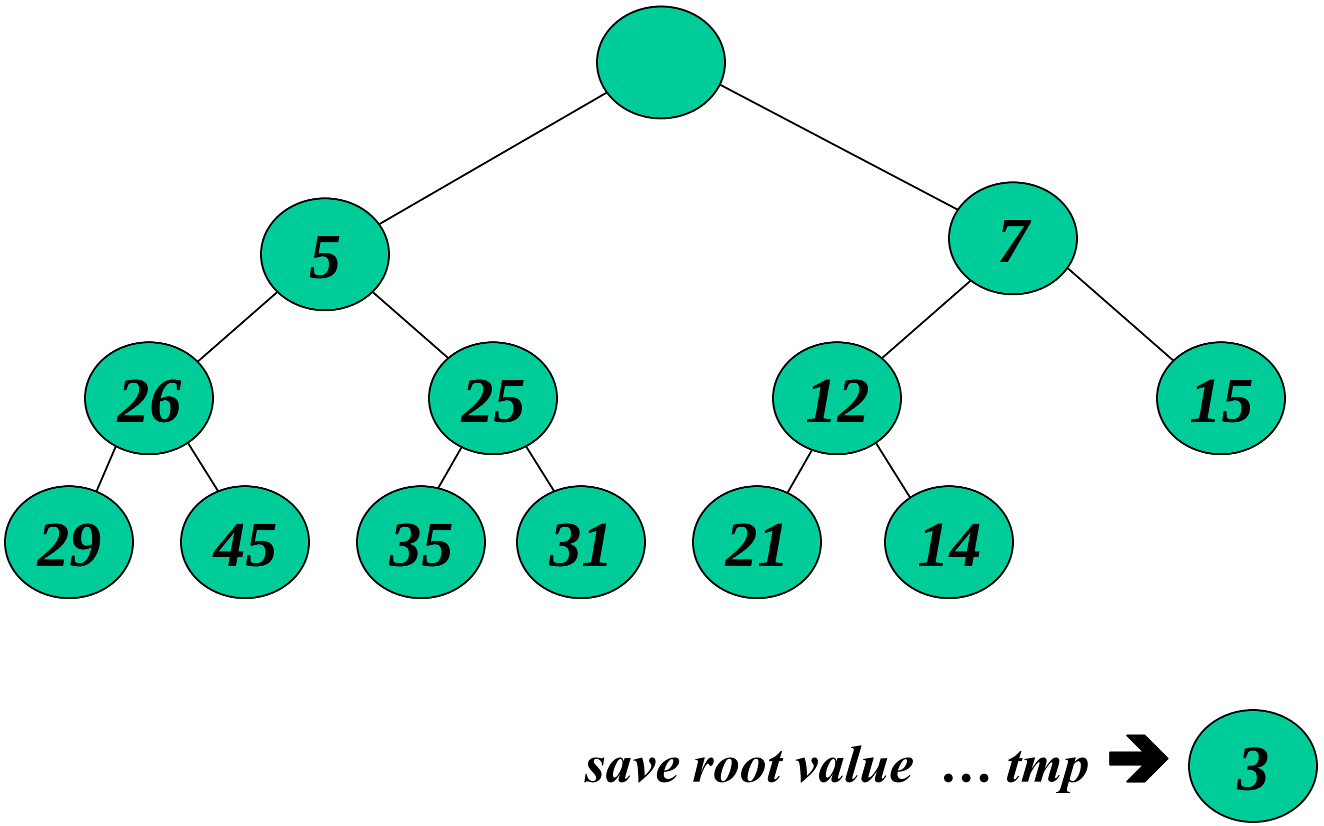
```
  while (child <= n) {
   /* find the larger child of the current
   parent */
   if ((child < n)&&
        (heap[child].key<heap[child+1].key))
        child++;
   if (temp.key >= heap[child].key) break;
   /* move to the next lower level */
   heap[parent] = heap[child];
   parent = child
   child *= 2;
 }
 heap[parent] = temp;
 return item;
}
```

Deleting a Value (note new tree!)



Delete 3

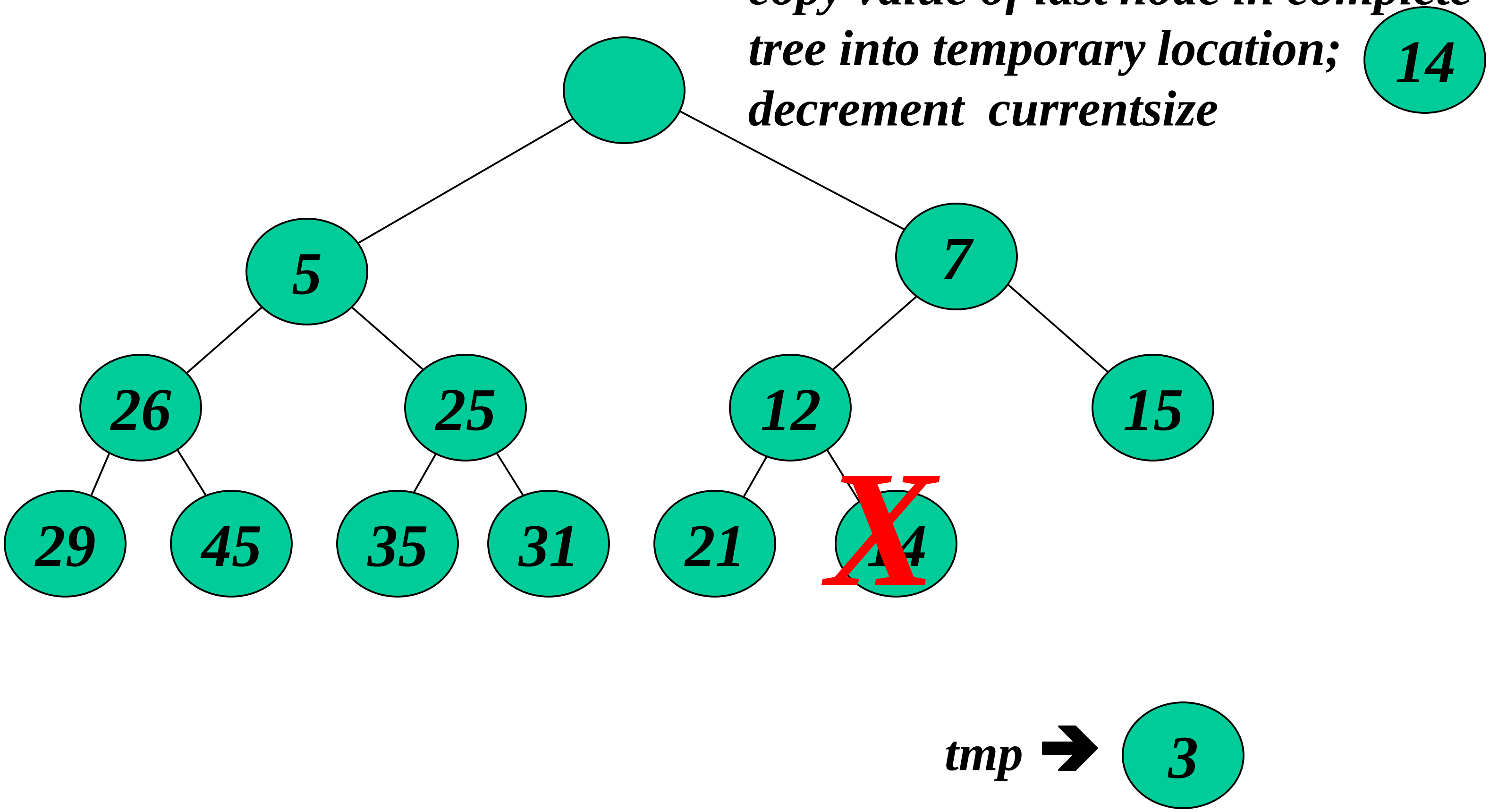
Deleting a Value



Delete 3

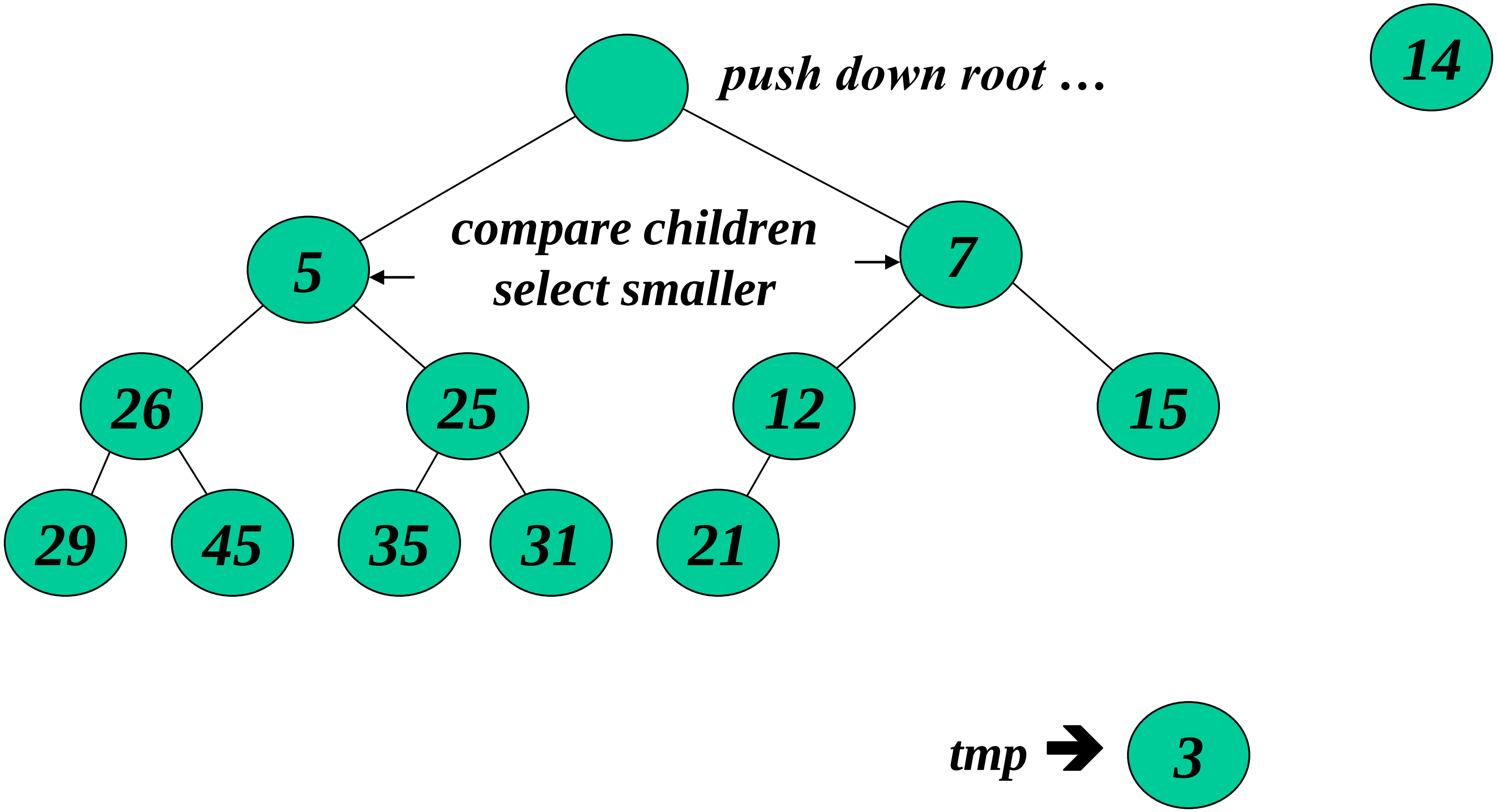
Deleting a Value

copy value of last node in complete tree into temporary location; decrement currentsize



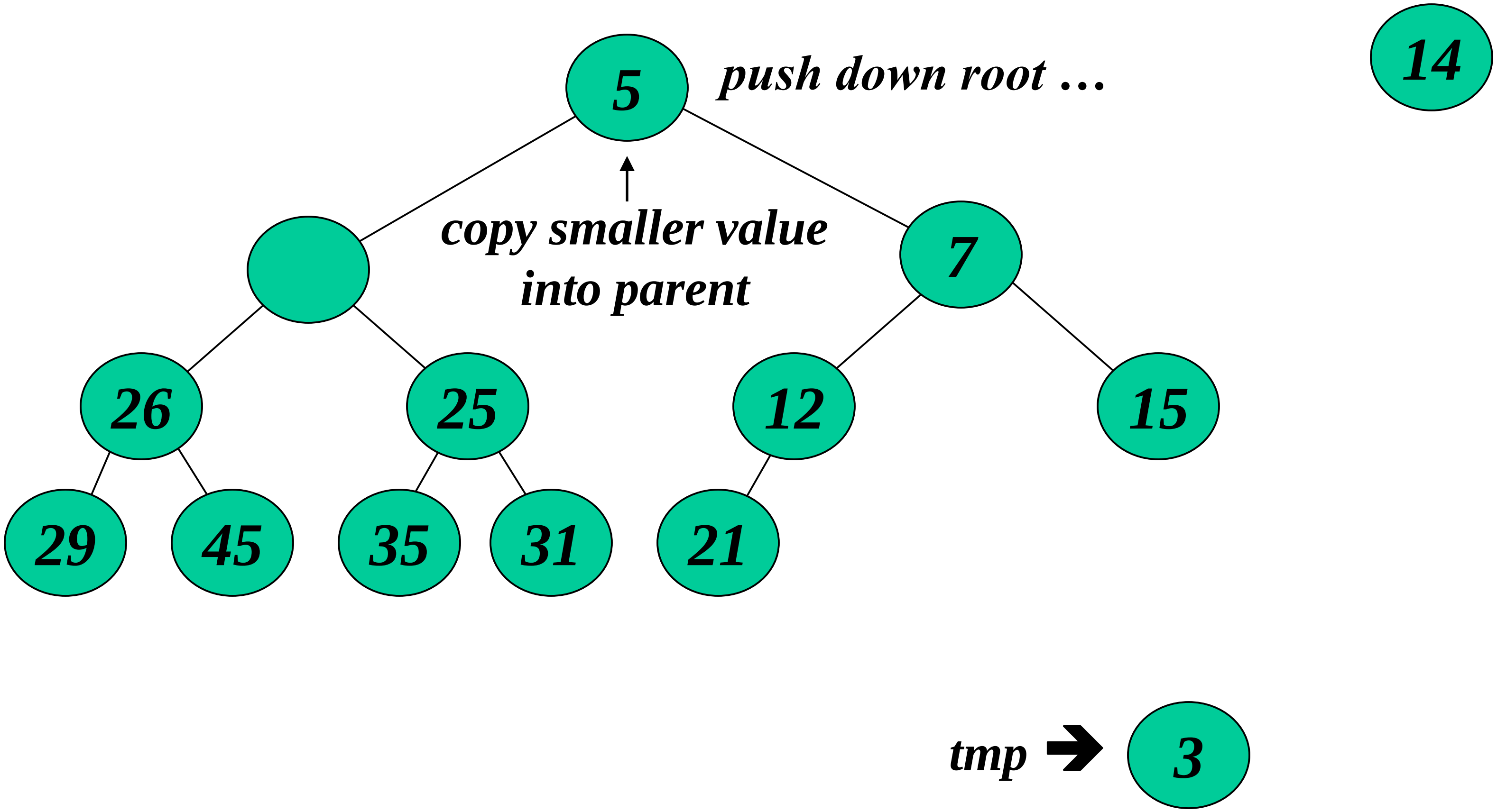
Delete 3

Deleting a Value



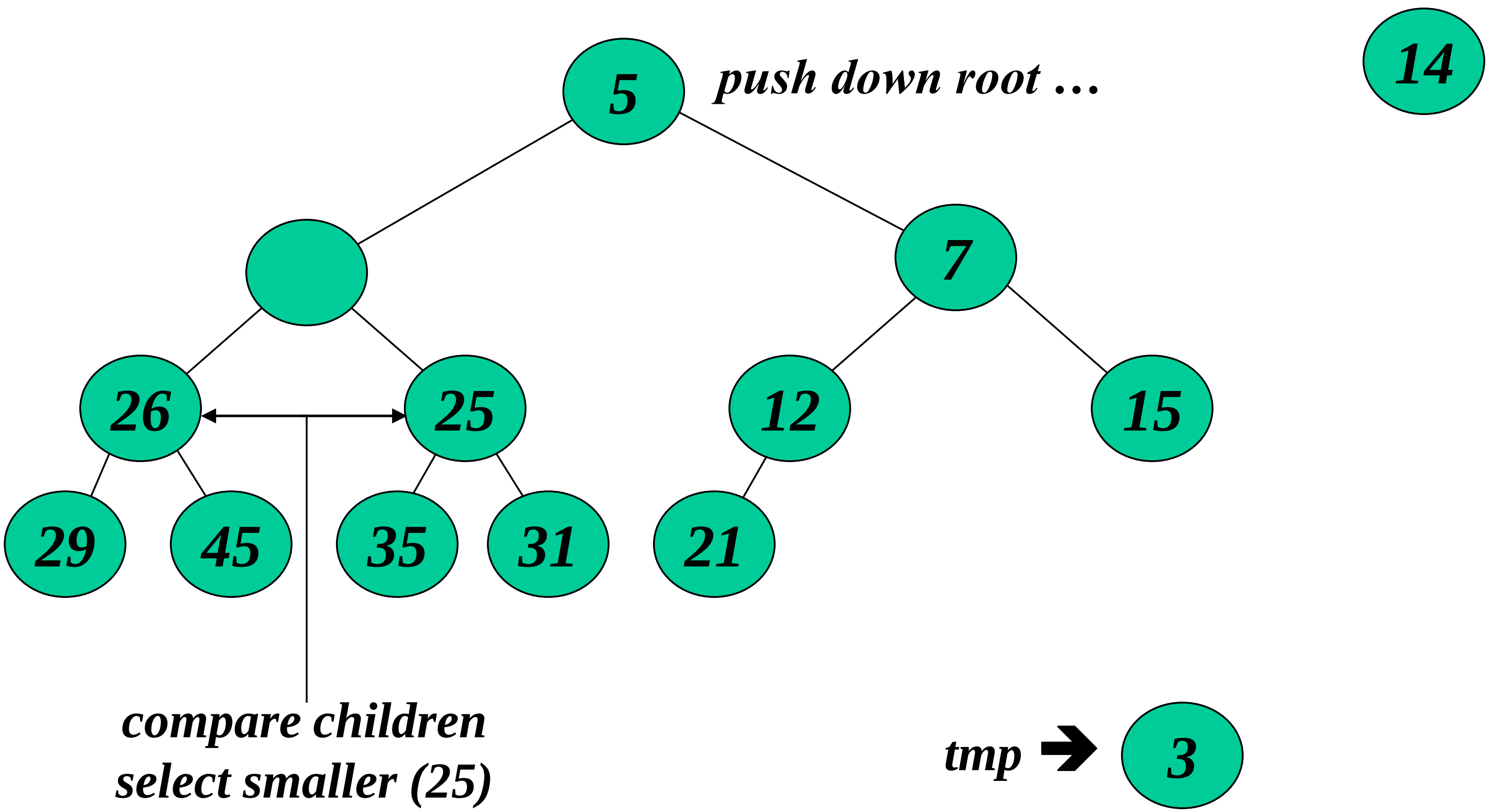
Delete 3

Deleting a Value



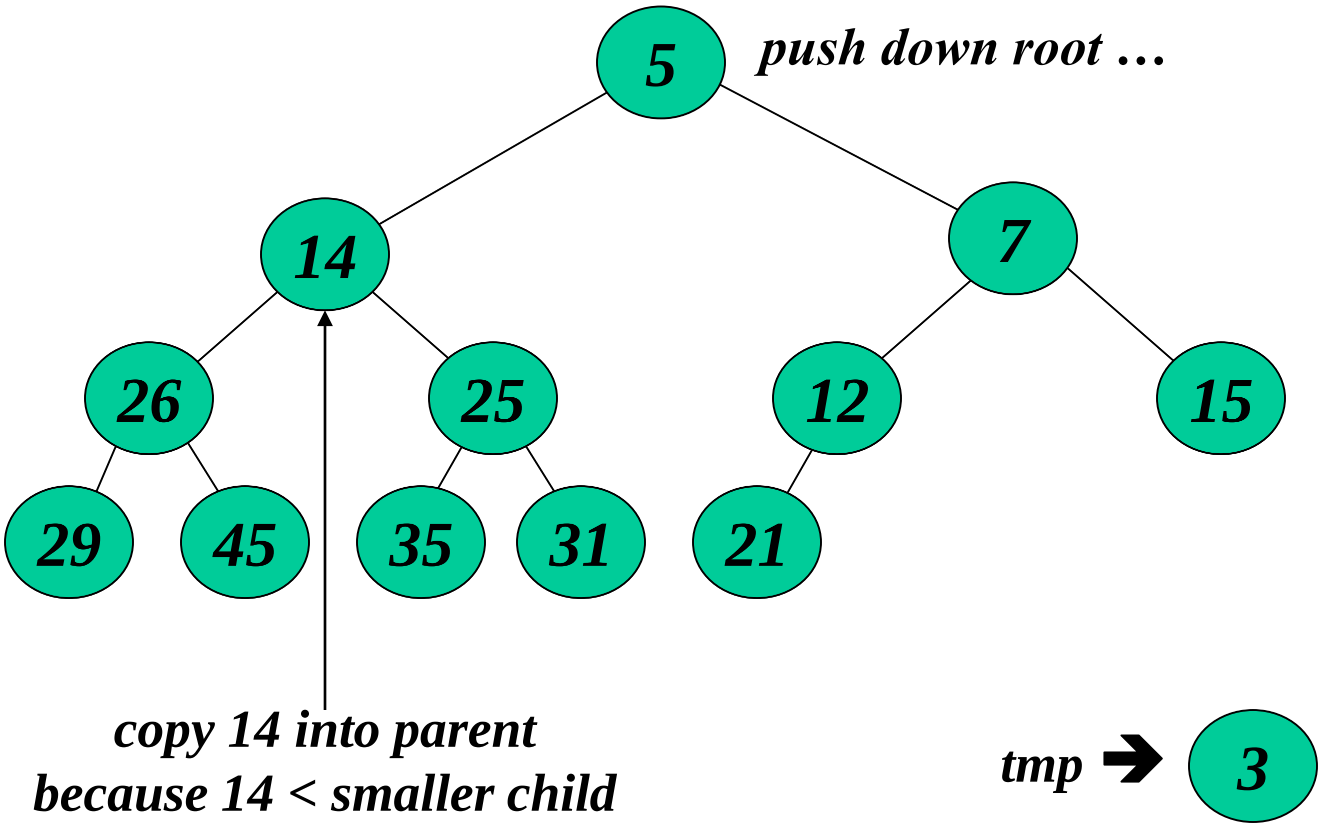
Delete 3

Deleting a Value



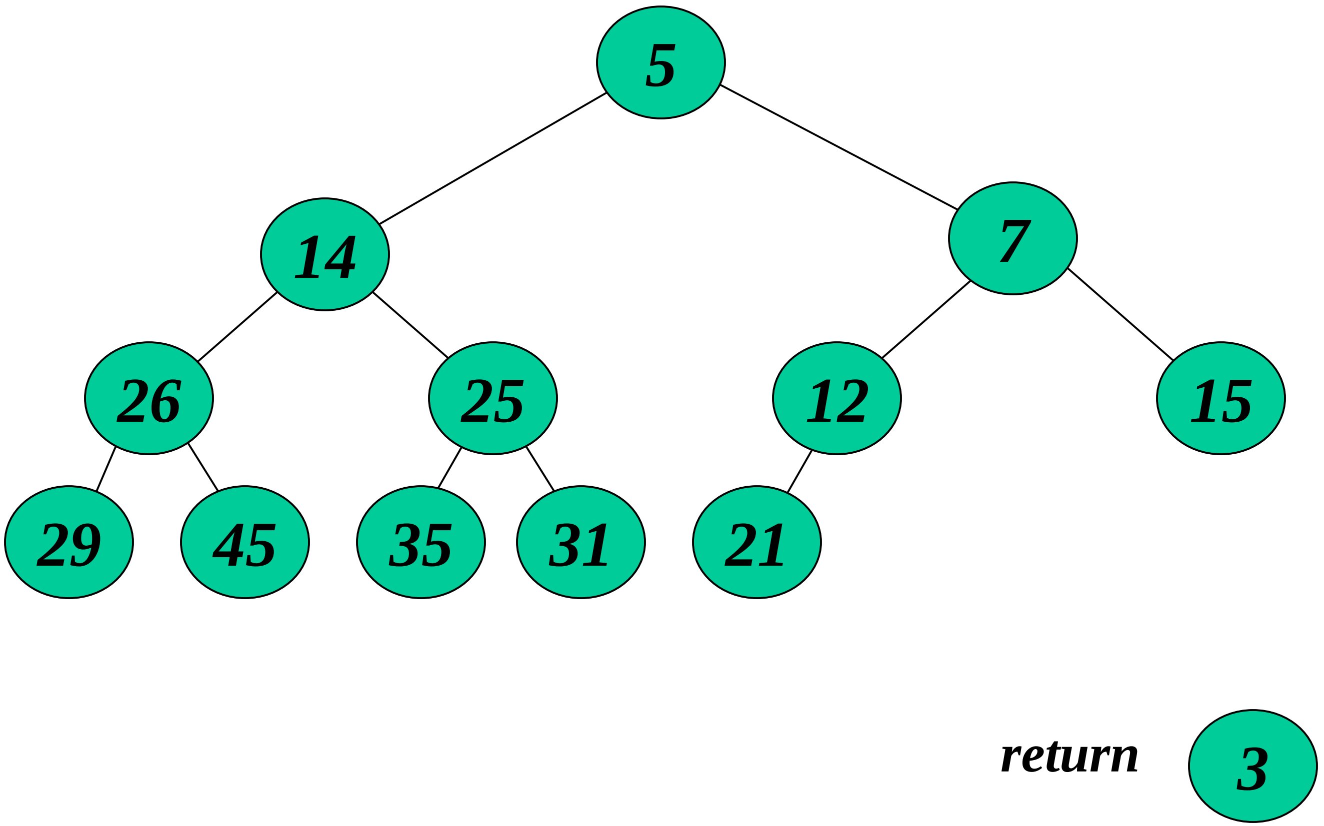
Delete 3

Deleting a Value



Delete 3

Deleting a Value



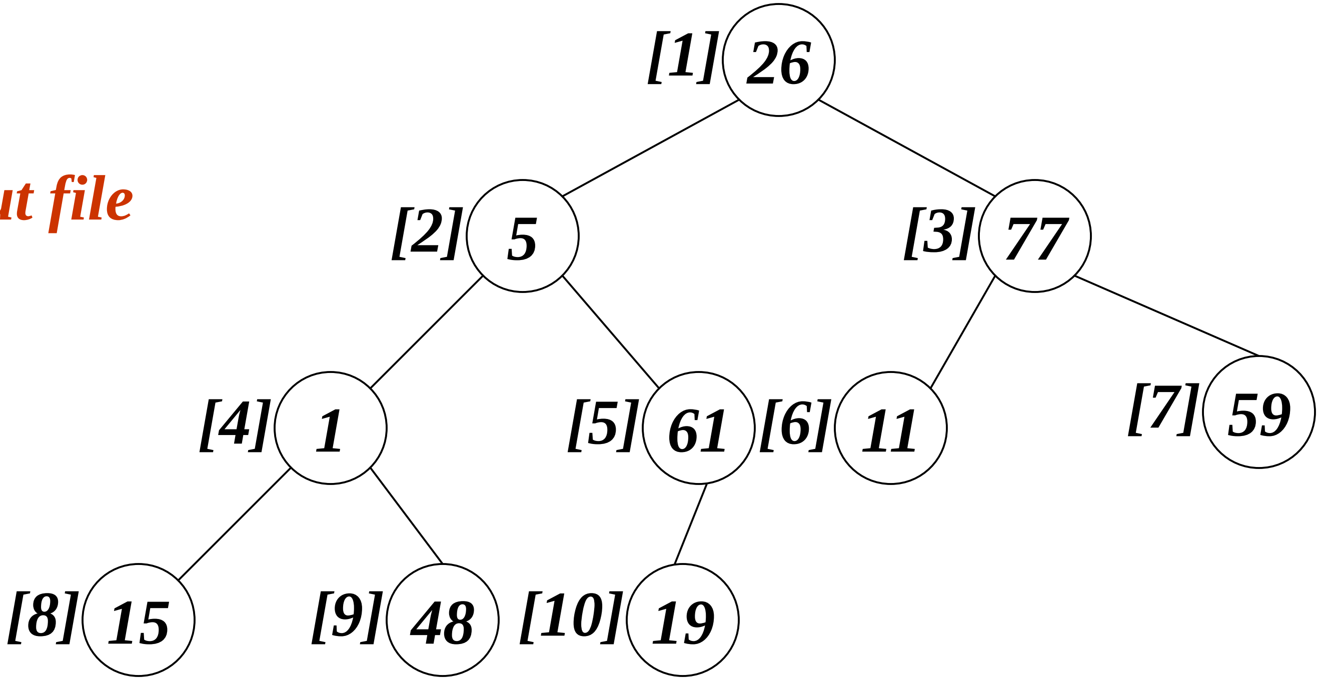
Delete 3

Application On Sorting: Heap Sort

- See an illustration first
  - Array interpreted as a binary tree

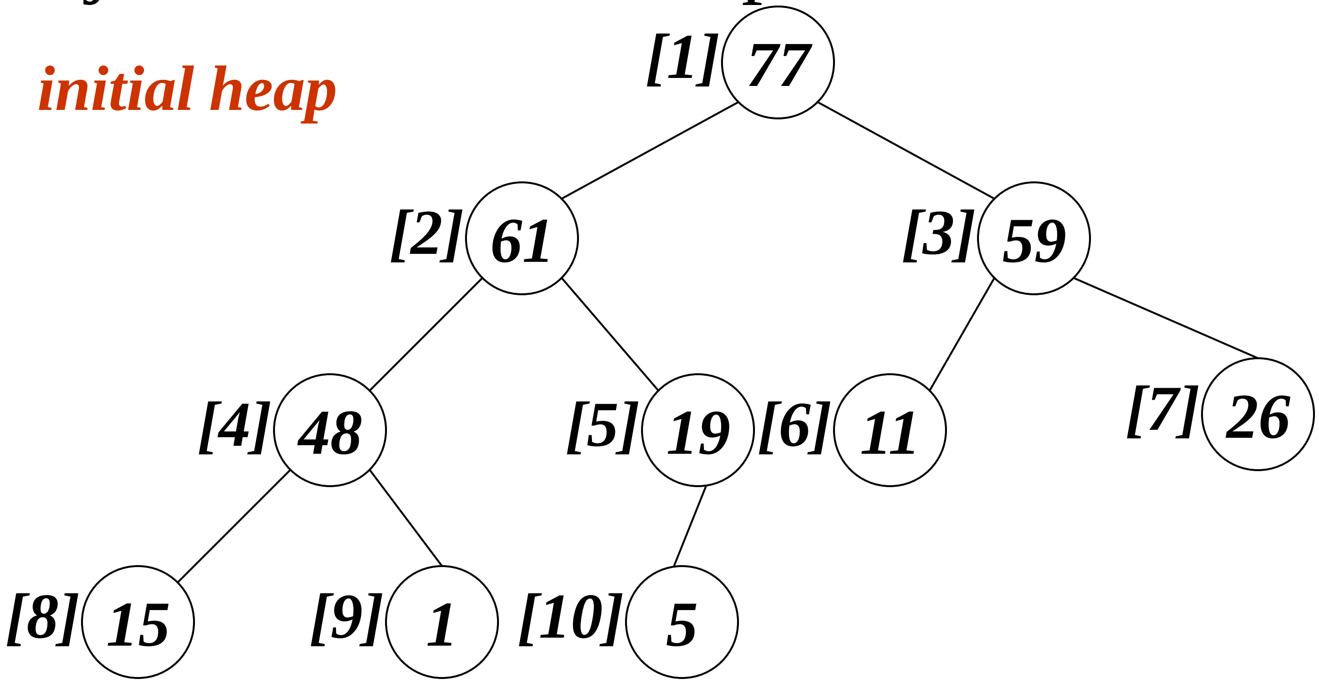
1	2	3	4	5	6	7	8	9	10
26	5	77	1	61	11	59	15	48	19

input file



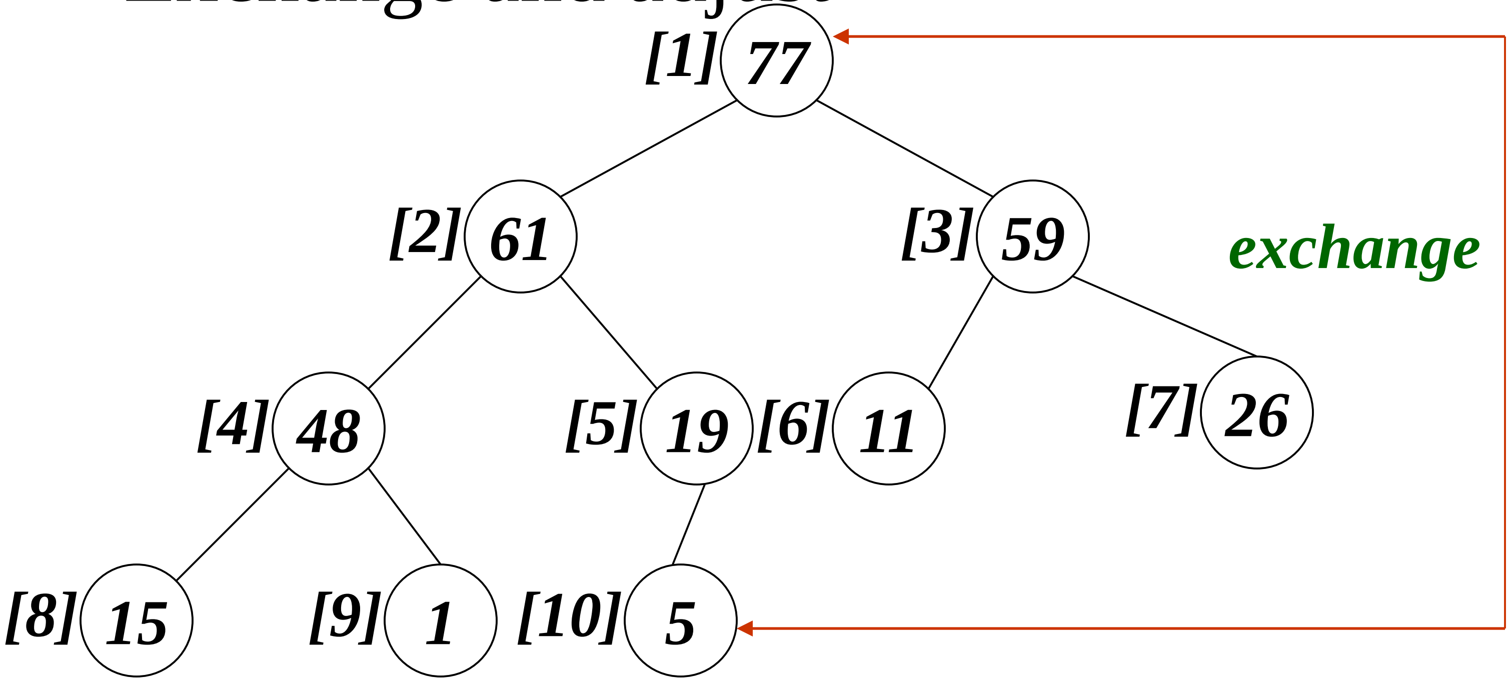
Heap Sort

- Adjust it to a MaxHeap

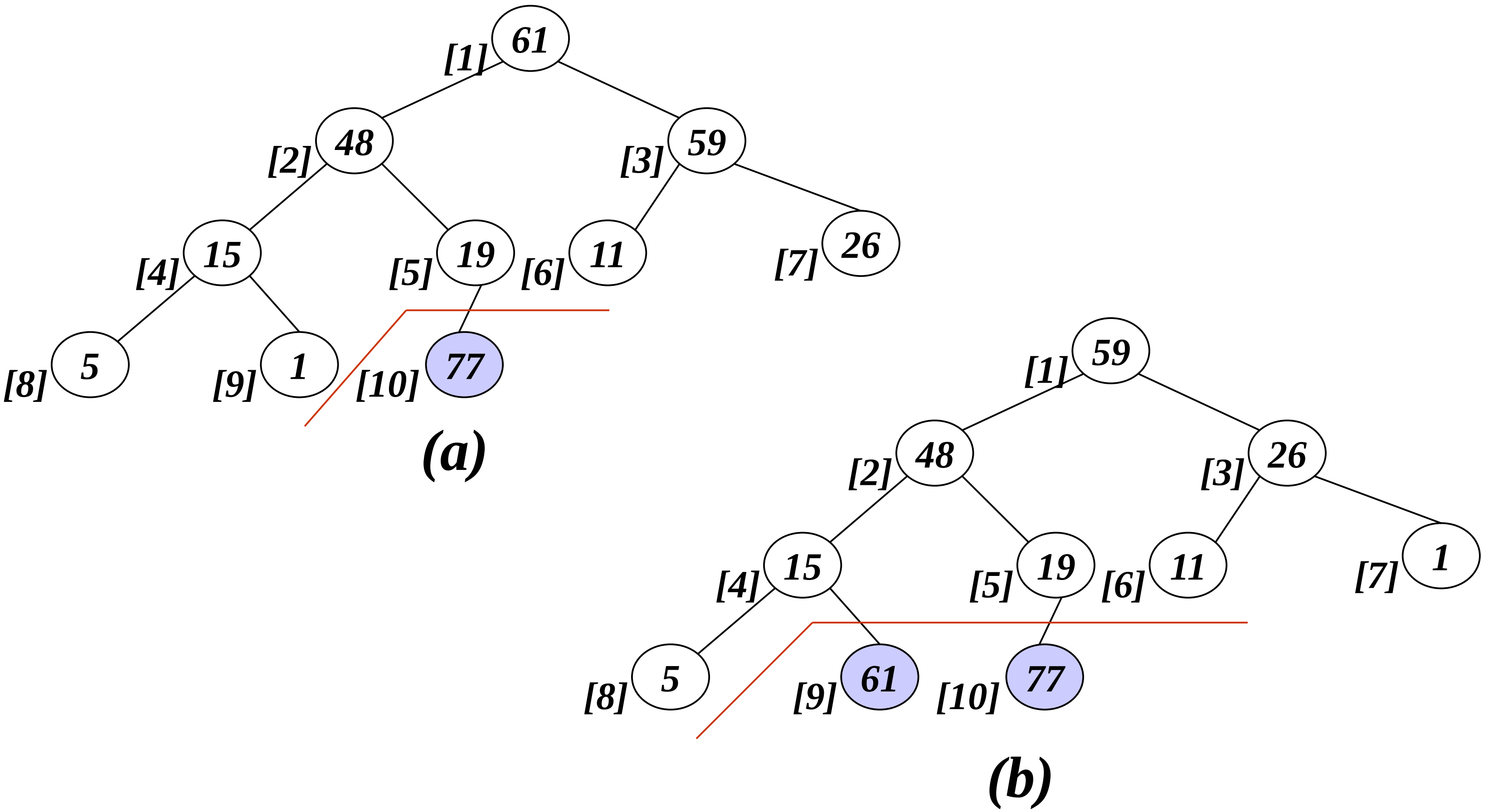


Heap Sort

- Exchange and adjust

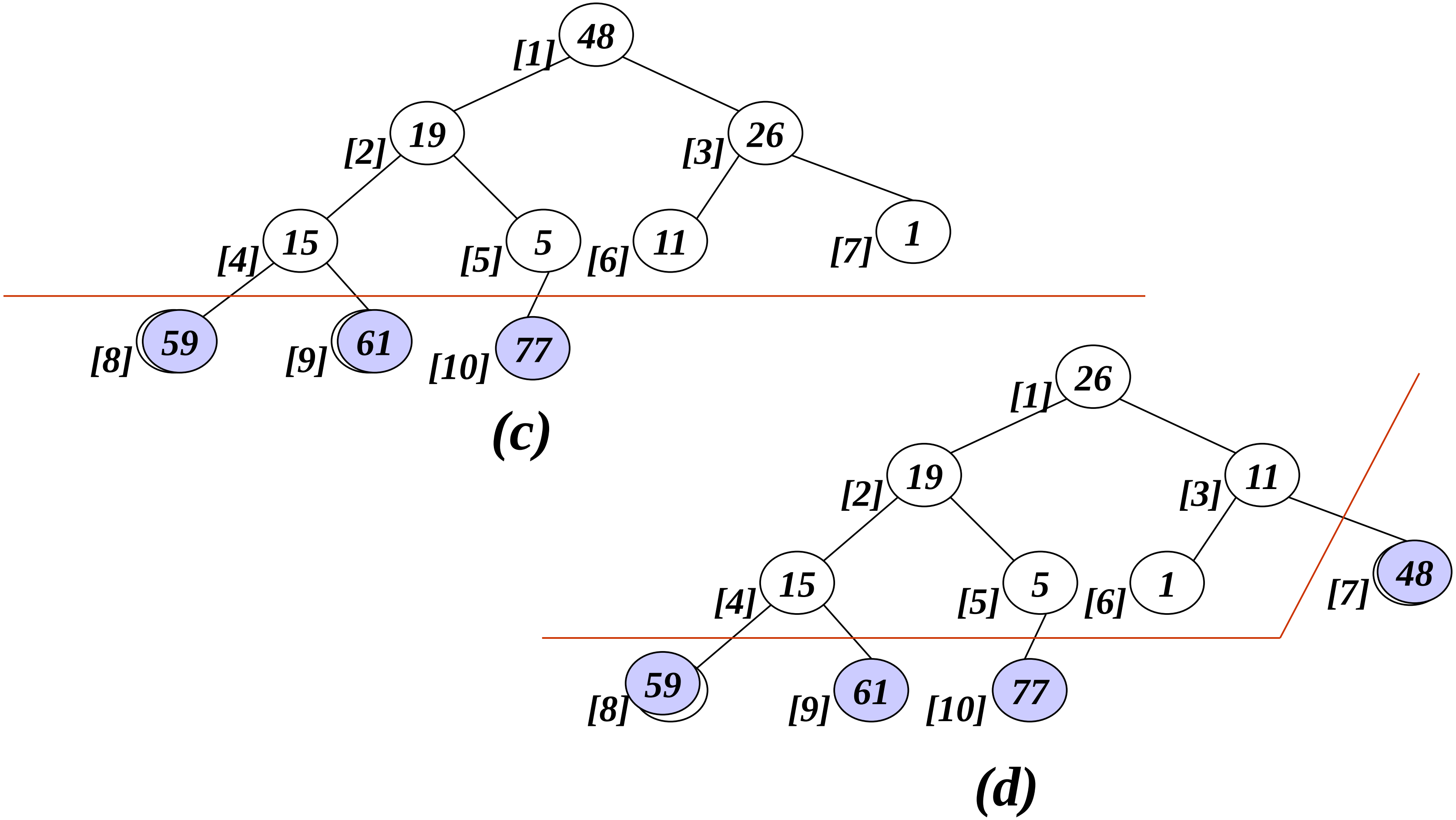


Heap Sort



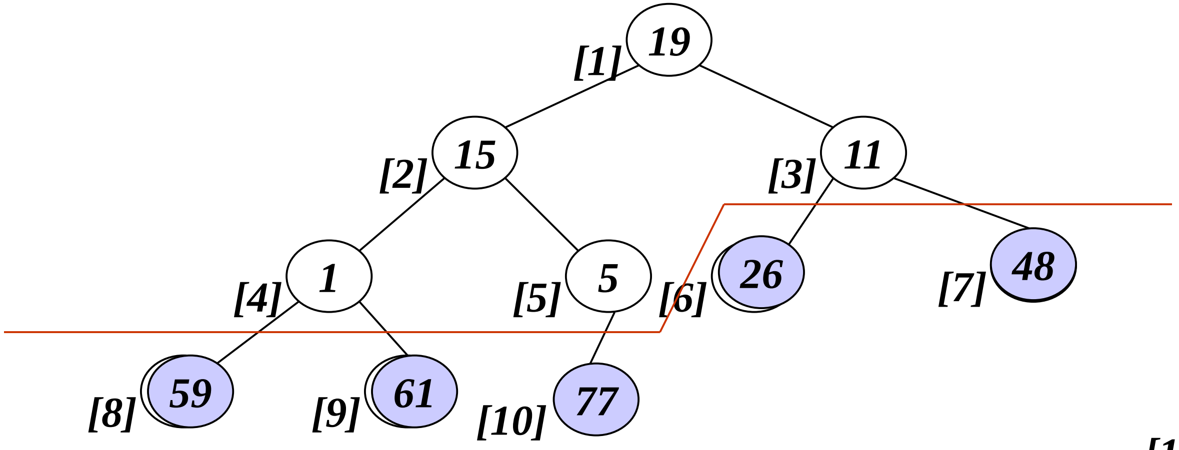
61
[1]
48
[2]
59
[3]
15
[4]
19
[5]
11
[6]
26
[7]
5
[8]
1
[9]
77
[10]
(a)
59
[1]
48
[2]
26
[3]
15
[4]
19
[5]
11
[6]
1
[7]
5
[8]
61
[9]
77
[10]
(b)

Heap Sort

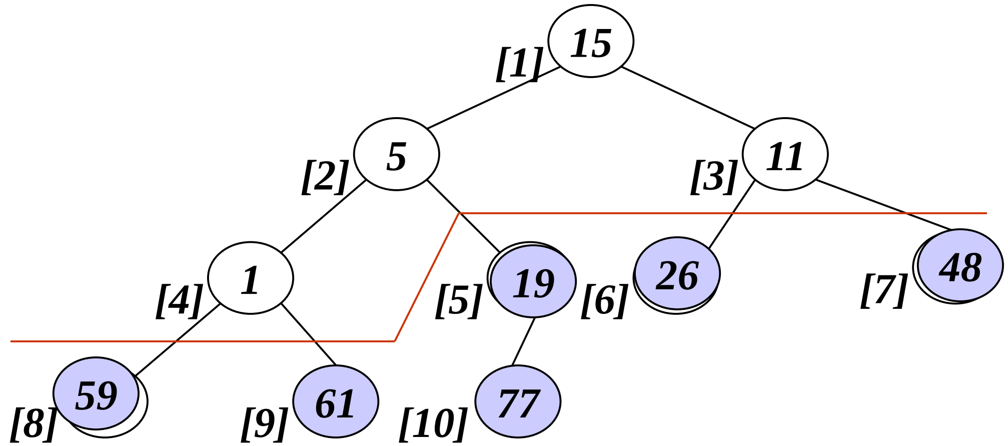


48
[1]
19
[2]
26
[3]
15
[4]
5
[5]
11
[6]
1
[7]
59
[8]
61
[9]
77
[10]
(c)
26
[1]
19
[2]
11
[3]
15
[4]
5
[5]
1
[6]
48
[7]
59
[8]
61
[9]
77
[10]
(d)

Heap Sort

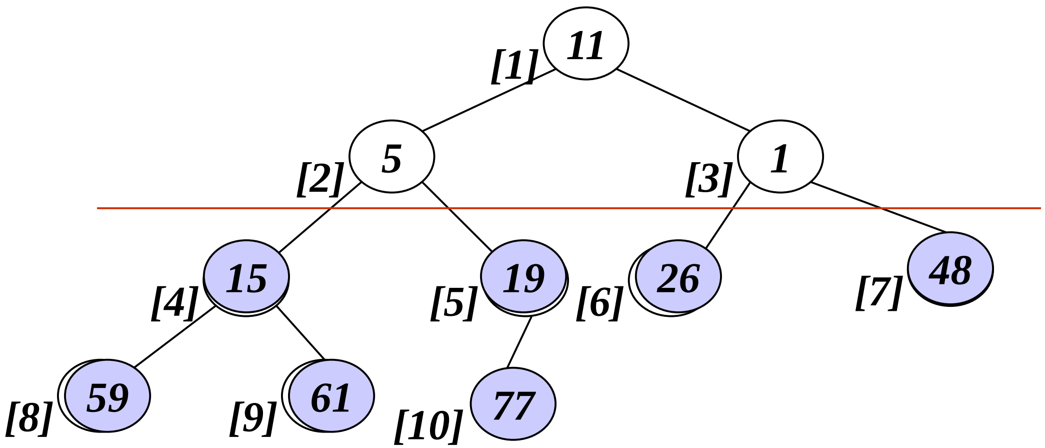

19
[1]
15
[2]
11
[3]
1
[4]
5
[5]
26
[6]
48
[7]
59
[8]
61
[9]
77
[10]

(e)


15
[1]
5
[2]
11
[3]
1
[4]
19
[5]
26
[6]
48
[7]
59
[8]
61
[9]
77
[10]

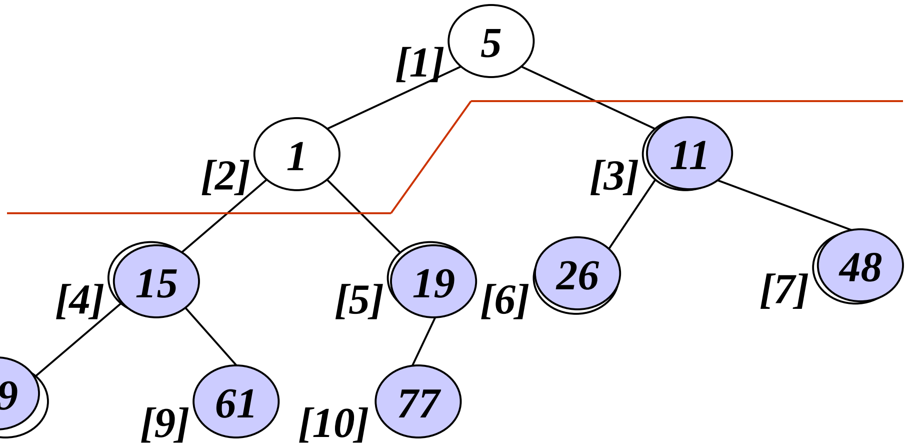
(f)

Heap Sort



[1] 11
[2] 5
[3] 1
[4] 15
[5] 19
[6] 26
[7] 48
[8] 59
[9] 61
[10] 77

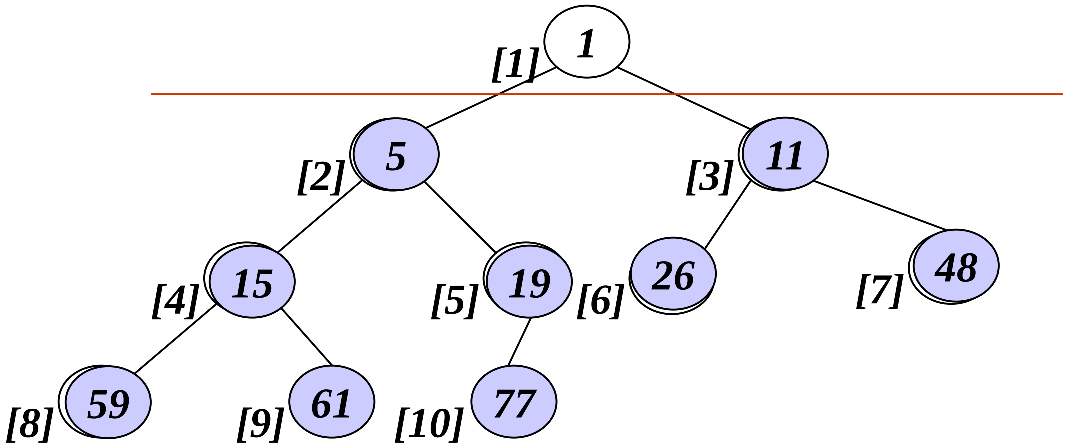
(g)



[1] 5
[2] 1
[3] 11
[4] 15
[5] 19
[6] 26
[7] 48
[8] 59
[9] 61
[10] 77

(h)

Heap Sort



(i)

- So results

77 61 59 48 26 19 15 11 5 1